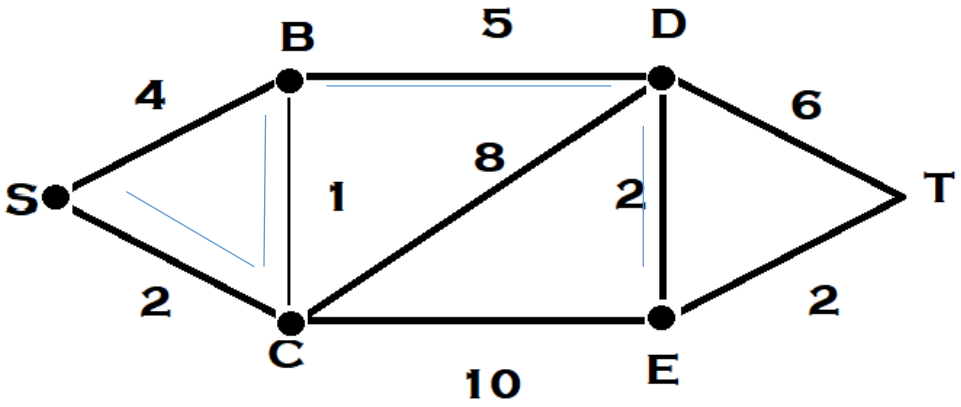

VERTICE	PASO 1	PASO 2	PASO 3	PASO 4	PASO 5	PASO 6
S	(0,S)	+	+	+	+	+
B	(4,S)	(3,C)	(3,C)	+	+	+
C	(2,S)	(2,S)	+	+	+	+
D	-------	(10,C)	(8,B)	(8,B)	*	*
E	-------	(12,C)	(12,C)	(10,D)	(10,D)	*
T	-------	--------	---------	(14,D)	(13,E)	(13,E)

Con el algoritmo de Dijkstra, se puede encontrar la ruta más corta o el camino más corto entre los nodos de un grafo. Específicamente, se puede encontrar el camino más corto desde un nodo (llamado el nodo de origen) a todos los otros nodos del grafo, generando un árbol del camino más corto.

Este algoritmo es usado por los dispositivos GPS para encontrar el camino más corto entre la ubicación actual y el destino del usuario. Tiene amplias aplicaciones en la industria, especialmente en aquellas áreas que requieren modelar redes.

Fue creado y publicado por el Dr. Edsger W. Dijkstra, un científico Neerlandés especialista en ciencias de la computación e ingeniería de software.

En 1959, publicó un artículo de tres páginas titulado: "A note on two problems in connexion with graphs", lo cual se traduce a español como "Una nota sobre dos problemas relacionados con grafos." En este artículo explicó su nuevo algoritmo.

Ejemplo

```
/*
EJEMPLO DE INPUT
5 9
```

```
1 2 7

1 4 2

2 3 1

2 4 2

3 5 4

4 2 3

4 3 8

4 5 5

5 3 5

1

*/

#include <stdio.h>

#include <vector>

#include <queue>

using namespace std;

#define MAX 10005 //maximo numero de vértices

#define Node pair< int , int > //definimos el nodo como un par( first , second ) donde first es el vertice adyacente y second el peso de la arista

#define INF 1<<30 //definimos un valor grande que represente la distancia infinita inicial, basta conque sea superior al maximo valor del peso en alguna de las aristas


//La cola de prioridad de C++ por default es un max-Heap (elemento de mayor valor en el tope)

//por ello es necesario realizar nuestro comparador para que sea un min-Heap

struct cmp {

   bool operator() ( const Node &a , const Node &b ) {

      return a.second > b.second;

   }

};

vector< Node > ady[ MAX ]; //lista de adyacencia

int distancia[ MAX ];     //distancia[ u ] distancia de vértice inicial a vértice con ID = u

bool visitado[ MAX ];     //para vértices visitados
```

```
priority_queue< Node , vector<Node> , cmp > Q; //priority queue propia del c++, usamos el comparador definido para que el de menor valor este en el tope

int V;                  //numero de vertices

int previo[ MAX ];      //para la impresion de caminos


//función de inicialización

void init(){

   for( int i = 0 ; i <= V ; ++i ){

      distancia[ i ] = INF;  //inicializamos todas las distancias con valor infinito

      visitado[ i ] = false; //inicializamos todos los vértices como no visitados

      previo[ i ] = -1;     //inicializamos el previo del vertice i con -1

   }

}


//Paso de relajacion

void relajacion( int actual , int adyacente , int peso ){

   //Si la distancia del origen al vertice actual + peso de su arista es menor a la distancia del origen al vertice adyacente

   if( distancia[ actual ] + peso < distancia[ adyacente ] ){

      distancia[ adyacente ] = distancia[ actual ] + peso;  //relajamos el vertice actualizando la distancia

      previo[ adyacente ] = actual;                    //a su vez actualizamos el vertice previo

      Q.push( Node( adyacente , distancia[ adyacente ] ) ); //agregamos adyacente a la cola de prioridad

   }

}


//Impresion del camino mas corto desde el vertice inicial y final ingresados

void print( int destino ){

   if( previo[ destino ] != -1 )    //si aun poseo un vertice previo

      print( previo[ destino ] );  //recursivamente sigo explorando

   printf("%d " , destino );       //terminada la recursion imprimo los vertices recorridos
```

```
}


void dijkstra( int inicial ){
   init(); //inicializamos nuestros arreglos
   Q.push( Node( inicial , 0 ) ); //Insertamos el vértice inicial en la Cola de Prioridad
   distancia[ inicial ] = 0;     //Este paso es importante, inicializamos la distancia del inicial como
0
   int actual , adyacente , peso;
   while( !Q.empty() ){                //Mientras cola no este vacia
      actual = Q.top().first;          //Obtengo de la cola el nodo con menor peso, en un comienzo
será el inicial
      Q.pop();                         //Sacamos el elemento de la cola
      if( visitado[ actual ] ) continue; //Si el vértice actual ya fue visitado entonces sigo sacando
elementos de la cola
      visitado[ actual ] = true;        //Marco como visitado el vértice actual


      for( int i = 0 ; i < ady[ actual ].size() ; ++i ){ //reviso sus adyacentes del vertice actual
         adyacente = ady[ actual ][ i ].first;   //id del vertice adyacente
         peso = ady[ actual ][ i ].second;       //peso de la arista que une actual con adyacente (
actual , adyacente )
         if( !visitado[ adyacente ] ){       //si el vertice adyacente no fue visitado
            relajacion( actual , adyacente , peso ); //realizamos el paso de relajacion
         }
      }
   }



   printf( "Distancias mas cortas iniciando en vertice %d\n" , inicial );
   for( int i = 1 ; i <= V ; ++i ){
      printf("Vertice %d , distancia mas corta = %d\n" , i , distancia[ i ] );
   }
```

```
    puts("\n**************Impresion de camino mas corto**************");
    printf("Ingrese vertice destino: ");
    int destino;
    scanf("%d" , &destino );
    print( destino );
    printf("\n");
}



int main(){
    int E , origen, destino , peso , inicial;
    scanf("%d %d" , &V , &E );
    while( E-- ){
        scanf("%d %d %d" , &origen , &destino , &peso );
        ady[ origen ].push_back( Node( destino , peso ) ); //consideremos grafo dirigido
        ady[ destino ].push_back( Node( origen , peso ) ); //grafo no dirigido
    }
    printf("Ingrese el vertice inicial: ");
    scanf("%d" , &inicial );
    dijkstra( inicial );
    return 0;
}
```